JavaScript Function Type

**Summary**: in this tutorial, you’ll learn about the JavaScript Function type, which is the type of all functions in JavaScript.

Introduction to the JavaScript Function type

In JavaScript, all [functions](https://www.javascripttutorial.net/javascript-function/) are [objects](https://www.javascripttutorial.net/javascript-objects/). They are the instances of the Function type. Because functions are objects, they have properties and methods like other objects.

Functions properties

Each function has two important properties: length and prototype.

* The length property determines the number of named arguments specified in the function declaration.
* The prototype property references the actual function object.

See the following example:

function add(x, y) {

return x + y;

}

console.log(add.length); *// 2*

console.log(add.prototype); *// Object{}*

Code language: JavaScript (javascript)

The add() function accepts two arguments x and y. Therefore, the length property returns two.

new.target

Typically, you call a function normally like this:

let result = add(10,20);

console.log(result); *// 30*

Code language: JavaScript (javascript)

Also, you can call a function with new keyword as a constructor:

let obj = new add(10,20);

Code language: JavaScript (javascript)

ES6 introduced the [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) pseudo-property that allows you to detect whether a function or constructor was called using the new operator.

If a function is called normally, the new.target is undefined. However, if the function is called using the new keyword as a constructor, the new.target return a reference to the constructor.

For example:

function add(x, y) {

console.log(new.target);

return x + y;

}

let result = add(10, 20);

let obj = new add(10, 20);

Code language: JavaScript (javascript)

Output:

undefined

[Function: add]

Code language: JavaScript (javascript)

By using the new.target, you can control how a function will be called.

For example, to prevent the add() function from being called with the new keyword as a constructor, you can throw an error by checking the new.target like this:

function add(x, y) {

if (new.target) {

throw 'The add function cannot be called as a constructor';

}

return x + y;

}

let obj = new add(10, 20);

console.log(obj);

Code language: JavaScript (javascript)

Function methods: apply, call, and bind

A function object has three important methods: [apply()](https://www.javascripttutorial.net/javascript-apply-method/), [call()](https://www.javascripttutorial.net/javascript-call/) and [bind()](https://www.javascripttutorial.net/javascript-bind/).

The apply() and call() methods

The apply() and call() methods call a function with a given this value and arguments.

The difference between the apply() and call() is that you need to pass the arguments to the apply() method as an array-like object, whereas you pass the arguments to the call() function individually. For example:

let cat = { type: 'Cat', sound: 'Meow' };

let dog = { type: 'Dog', sound: 'Woof' };

const say = function (message) {

console.log(message);

console.log(this.type + ' says ' + this.sound);

};

say.apply(cat, ['What does a cat say?']);

say.apply(dog, ['What does a dog say?']);

Code language: JavaScript (javascript)

Output:

What does a cat sound?

Cat says Meow

What does a dog sound?

Dog says Woof

In this example:

First, declare two objects cat and dog with two properties:

let cat = { type: 'Cat', sound: 'Meow' };

let dog = { type: 'Dog', sound: 'Woof' };

Code language: JavaScript (javascript)

Second, define the say() function that accepts one argument:

const say = function (message) {

console.log(message);

console.log(this.type + ' says ' + this.sound);

};

Code language: JavaScript (javascript)

Third, call the say() function via the apply() method:

say.apply(cat, ['What does a cat say?']);

Code language: CSS (css)

In this example, the first argument of the  apply() method is the cat object. Therefore, the this object in the say() function references the cat object.

Fourth, call say() function and pass the dog object:

say.apply(dog, ['What does a dog say?']);

Code language: CSS (css)

In this example, the this in the say() function reference the dog object.

The call() method like the apply() method except for the way you pass the arguments to the function:

say.call(cat, 'What does a cat say?');

say.call(dog, 'What does a dog say?');

Code language: JavaScript (javascript)

The bind() method

The bind() method creates a new function instance whose this value is bound to the object that you provide. For example:

First, define an object named car:

let car = {

speed: 5,

start: function() {

console.log('Start with ' + this.speed + ' km/h');

}

};

Code language: JavaScript (javascript)

Then, define another object named aircraft:

let aircraft = {

speed: 10,

fly: function() {

console.log('Flying');

}

};

Code language: JavaScript (javascript)

The aircraft has no start() method. To start an aircraft, you can use the bind() method of the start() method of the car object:

let taxiing = car.start.bind(aircraft);

Code language: JavaScript (javascript)

In this statement, we change the this value inside the start() method of the car object to the  aircraft object.  The bind() method returns a new function that is assigned to the taxiing variable.

Now, you can call the start() method via the taxiing variable:

taxiing();

It will show the following message:

Start with 10 km/h

Code language: JavaScript (javascript)

The following uses the call() method to call the start() method on the aircraft object:

car.start.call(aircraft);

Code language: CSS (css)

As you can see, the bind() method creates a new function that you can execute later while the call() method executes the function immediately. This is the main difference between the bind() and call() methods.

Technically, the aircraft object borrows the start() method of the car object via the bind(), call() or apply() method.

For this reason, the bind(), call(), and apply() methods are also known as borrowing functions.

Summary

* All functions are instances of the Function type, which are the objects that have properties and methods.
* A function has two important properties: length and prototype.
* A function also has three important methods: call(), apply(), and bind().

JavaScript call() Method

**Summary**: in this tutorial, you will learn about the JavaScript call() method and how to use it more effectively.

Introduction to the JavaScript call() method

In JavaScript, a [function](https://www.javascripttutorial.net/javascript-function/) is an instance of the [Function](https://www.javascripttutorial.net/javascript-function-type/) type. For example:

function add(x, y) {

return x + y;

}

console.log(add instanceof Function); *// true*

Code language: JavaScript (javascript)

The Function.prototype type has the call() method with the following syntax:

functionName.call(thisArg, arg1, arg2, ...);

Code language: JavaScript (javascript)

In this syntax, the call() method calls a function functionName with the arguments (arg1, arg2, …) and the this set to thisArg object inside the function.

* The thisArg is the object that the this object references inside the function functionName.
* The arg1, arg2, .. are the function arguments passed into the functionName.

The call() method returns the result of calling the functionName().

The following example defines the add() function and calls it normally:

function add(x, y) {

return x + y;

}

let result = add(10, 20);

console.log(result); *// 30*

Code language: JavaScript (javascript)

The following calls the add() function but use the call() method instead:

function add(x, y) {

return x + y;

}

let result = add.call(this, 10, 20);

console.log(result); *// 30*

Code language: JavaScript (javascript)

By default, the [this](https://www.javascripttutorial.net/javascript-this/) inside the function is set to the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/) i.e., window in the web browsers and global in Node.js.

Note that in the strict mode, the this inside the function is set to undefined instead of the global object.

Consider the following example:

var greeting = 'Hi';

var messenger = {

greeting: 'Hello'

}

function say(name) {

console.log(this.greeting + ' ' + name);

}

Code language: JavaScript (javascript)

Inside the say() function, we reference the greeting via the this value. If you just invoke the say() function via the call() method as follows:

say.call(this,'John');

Code language: JavaScript (javascript)

It’ll show the following output to the console:

"Hi John"

Code language: JavaScript (javascript)

However, when you invoke the call() method of say function object and pass the messenger object as the this value:

say.call(messenger,'John');

Code language: JavaScript (javascript)

The output will be:

"Hello John"

Code language: JavaScript (javascript)

In this case, the this value inside the say() function references the messenger object, not the global object.

Using the JavaScript call() method to chain constructors for an object

You can use the call() method for chaining constructors of an object. Consider the following example:

function Box(height, width) {

this.height = height;

this.width = width;

}

function Widget(height, width, color) {

Box.call(this, height, width);

this.color = color;

}

let widget = new Widget('red', 100, 200);

console.log(widget);

Code language: JavaScript (javascript)

Output:

Widget { height: 'red', width: 100, color: 200 }

Code language: JavaScript (javascript)

In this example:

* First, initialize the Box object with two properties: height and width.
* Second, invoke the call() method of the Box object inside the Widget object, set the this value to the Widget object.

Using the JavaScript call() method for function borrowing

The following example illustrates how to use the call() method for borrowing functions:

const car = {

name: 'car',

start() {

console.log('Start the ' + this.name);

},

speedUp() {

console.log('Speed up the ' + this.name);

},

stop() {

console.log('Stop the ' + this.name);

},

};

const aircraft = {

name: 'aircraft',

fly() {

console.log('Fly');

},

};

car.start.call(aircraft);

car.speedUp.call(aircraft);

aircraft.fly();

Code language: JavaScript (javascript)

Output:

Start the aircraft

Speed up the aircraft

Fly

Code language: JavaScript (javascript)

How it works.

First, define a car object with one property name and three methods start, speedUp, and stop:

const car = {

name: 'car',

start() {

console.log('Start the ' + this.name);

},

speedUp() {

console.log('Speed up the ' + this.name);

},

stop() {

console.log('Stop the ' + this.name);

},

};

Code language: JavaScript (javascript)

Second, define the aircraft object with one property name and a method:

const aircraft = {

name: 'aircraft',

fly() {

console.log('Fly');

},

};

Code language: JavaScript (javascript)

Third, call the start() and speedUp() method of the car object and the fly() method of the aircraft object. However, passing the aircraft as the first argument into the start() and speedUp() methods:

car.start.call(aircraft);

car.speedUp.call(aircraft);

aircraft.fly();

Code language: JavaScript (javascript)

Inside the start() and speedUp() methods, the this references the aircraft object, not the car object. Therefore, the this.name returns the 'aircraf' string. Hence, the methods output the following message:

Start the aircraft

Speed up the aircraft

Code language: plaintext (plaintext)

Technically, the aircraft object borrows the start() and speedUp() method of the car object. And function borrowing refers to an object that uses a method of another object.

The following example illustrates how the arguments object borrows the filter() method of the Array.prototype via the call() function:

function isOdd(number) {

return number % 2;

}

function getOddNumbers() {

return Array.prototype.filter.call(arguments, isOdd);

}

let results = getOddNumbers(10, 1, 3, 4, 8, 9);

console.log(results);

Code language: JavaScript (javascript)

Output:

[ 1, 3, 9 ]

Code language: JavaScript (javascript)

How it works.

First, define the isOdd() function that returns true if the number is an odd number:

function isOdd(number) {

return number % 2;

}

Code language: JavaScript (javascript)

Second, define the getOddNumbers() function that accepts any number of arguments and returns an array that contains only odd numbers:

function getOddNumbers() {

return Array.prototype.filter.call(arguments, isOdd);

}

Code language: JavaScript (javascript)

In this example, the arguments object borrows the filter() method of the Array.prototype object.

Third, call the getOddNumbers() function:

let results = getOddNumbers(10, 1, 3, 4, 8, 9);

console.log(results);

Code language: JavaScript (javascript)

In this tutorial, you have learned about the JavaScript call() method and how to use it more effectively.

JavaScript apply() method

**Summary**: in this tutorial, you’ll learn about the JavaScript apply() method of the Function type and how to use it effectively.

Introduction to the JavaScript apply() method

The Function.prototype.apply() method allows you to call a [function](https://www.javascripttutorial.net/javascript-function/) with a given [this](https://www.javascripttutorial.net/javascript-this/) value and arguments provided as an [array](https://www.javascripttutorial.net/javascript-array/). Here is the syntax of the apply() method:

fn.apply(thisArg, [args]);

Code language: JavaScript (javascript)

The apply() method accepts two arguments:

* The thisArg is the value of this provided for the call to the function fn.
* The args argument is an array that specifies the arguments of the function fn. Since the ES5, the args argument can be an array-like object or array object.

THe apply() method is similar to the [call()](https://www.javascripttutorial.net/javascript-call/) method except that it takes the arguments of the function as an array instead of the individual arguments.

JavaScript apply() method examples

Let’s take some examples of using the apply() method.

1) Simple JavaScript apply() method example

Suppose that you have a person object:

const person = {

firstName: 'John',

lastName: 'Doe'

}

Code language: JavaScript (javascript)

…and a function named greet() as follows:

function greet(greeting, message) {

return `${greeting} ${this.firstName}. ${message}`;

}

Code language: JavaScript (javascript)

The greet() function accepts two parameters: greeting and message. Inside the greet() function, we reference an object that has the firstName property.

The following example shows how to use the apply() method to call the greet() function with the this set to the person object:

let result = greet.apply(person, ['Hello', 'How are you?']);

console.log(result);

Code language: JavaScript (javascript)

Output:

Hello John. How are you?

Code language: JavaScript (javascript)

In this example, we set the this value inside the function to the person object. The arguments of the greet() function was passed into the apply() method as an array.

The apply() method invoked the greet() function with the this value set to the person object and arguments as an array ['Hello', 'How are you?'].

If you use the call() method, you need to pass the arguments of the greet() function separately as follows:

let result = greet.call(person, Hello', 'How are you?');

Code language: JavaScript (javascript)

2) Function borrowing

The apply() method allows an object to borrow the method of another object without duplicating the code.

Suppose that you have the following computer object:

const computer = {

name: 'MacBook',

isOn: false,

turnOn() {

this.isOn = true;

return `The ${this.name} is On`;

},

turnOff() {

this.isOn = false;

return `The ${this.name} is Off`;

}

};

Code language: JavaScript (javascript)

… and the following server object:

const server = {

name: 'Dell PowerEdge T30',

isOn: false

};

Code language: JavaScript (javascript)

The server object doesn’t have the turnOn() and turnOff() methods.

To execute the turnOn() method of the computer object on the server object, you can use the apply() method as follows:

let result = computer.turnOn.apply(server);

console.log(result);

Code language: JavaScript (javascript)

Output:

The Dell PowerEdge T30 is On

Code language: JavaScript (javascript)

In this example, the server object borrows the turnOn() method of the computer object.

Similarly, you can call the turnOff() method of the computer object on the server object:

let result = computer.turnOff.apply(server);

console.log(result);

Code language: JavaScript (javascript)

Output:

The Dell PowerEdge T30 is Off

Code language: JavaScript (javascript)

3) Using the apply() method to append an array to another

The apply() method allows you to append elements of an array to another:

let arr = [1, 2, 3];

let numbers = [4, 5, 6];

arr.push.apply(arr, numbers);

console.log(arr);

Code language: JavaScript (javascript)

In this example, the apply() method modifies the original array arr. Note that the [Array.prototype.concat()](https://www.javascripttutorial.net/javascript-array-concat/) method also provides the same result except that it returns the new array instead of modifying the original array.

Summary

* The apply() method invokes a function with a given this value and arguments provided as an array.
* The apply() method is similar to the call() method excepts that it accepts the arguments of the function as an array instead of individual arguments.

JavaScript bind() Method

**Summary**: in this tutorial, you will learn about the JavaScript bind() method and know how to use it effectively.

Introduction to JavaScript bind() method

The bind() method returns a new [function](https://www.javascripttutorial.net/javascript-function/), when invoked, has its [this](https://www.javascripttutorial.net/javascript-this/) sets to a specific value.

The following illustrates the syntax of the bind() method:

fn.bind(thisArg[, arg1[, arg2[, ...]]])

Code language: CSS (css)

In this syntax, the bind() method returns a copy of the function fn with the specific this value (thisArg) and arguments (arg1, arg2, …).

Unlike the [call()](https://www.javascripttutorial.net/javascript-call/) and [apply()](https://www.javascripttutorial.net/javascript-apply-method/) methods, the bind() method doesn’t immediately execute the function. It just returns a new version of the function whose this sets to thisArg argument.

Using JavaScript bind() for function binding

When you pass a method an [object](https://www.javascripttutorial.net/javascript-objects/) is to another function as a [callback](https://www.javascripttutorial.net/javascript-callback/), the this is lost. For example:

let person = {

name: 'John Doe',

getName: function() {

console.log(this.name);

}

};

setTimeout(person.getName, 1000);

Code language: JavaScript (javascript)

Output:

undefined

Code language: JavaScript (javascript)

As you can see clearly from the output, the person.getName() returns undefined instead of 'John Doe'.

This is because [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) received the function person.getName separately from the person object.

The statement:

setTimeout(person.getName, 1000);

Code language: CSS (css)

can be rewritten as:

let f = person.getName;

setTimeout(f, 1000); *// lost person context*

Code language: JavaScript (javascript)

The this inside the setTimeout() function is set to the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/) in non-strict mode and undefined in the strict mode.

Therefore, when the callback person.getName is invoked, the name does not exist in the global object, it is set to undefined.

To fix the issue, you can wrap the call to the person.getName method in an [anonymous function](https://www.javascripttutorial.net/javascript-anonymous-functions/), like this:

setTimeout(function () {

person.getName();

}, 1000);

Code language: JavaScript (javascript)

This works because it gets the person from the outer scope and then calls the method getName().

Or you can use the bind() method:

let f = person.getName.bind(person);

setTimeout(f, 1000);

Code language: JavaScript (javascript)

In this code:

* First, bind the person.getName method to the person object.
* Second, pass the bound function f with this value set to the person object to the setTimeout() function.

Using bind() to borrow methods from a different object

Suppose you have a runner object that has the run() method:

let runner = {

name: 'Runner',

run: function(speed) {

console.log(this.name + ' runs at ' + speed + ' mph.');

}

};

Code language: JavaScript (javascript)

And the flyer object that has the fly() method:

let flyer = {

name: 'Flyer',

fly: function(speed) {

console.log(this.name + ' flies at ' + speed + ' mph.');

}

};

Code language: JavaScript (javascript)

If you want the flyer object to be able to run, you can use the bind() method to create the run() function with the this  sets to the flyer object:

let run = runner.run.bind(flyer, 20);

run();

Code language: JavaScript (javascript)

In this statement:

* Call the bind() method of the runner.run() method and pass in the flyer object as the first argument and 20 as the second argument.
* Invoke the run() function.

Output:

Flyer runs at 20 mph.

The ability to borrow a method of an object without making a copy of that method and maintain it in two separate places is very powerful in JavaScript.

Summary

* The bind() method creates a new function, when invoked, has the this sets to a provided value.
* The bind() method allows an object to borrow a method from another object without making a copy of that method. This is known as function borrowing in JavaScript.

JavaScript Closures

**Summary**: in this tutorial, you will learn about JavaScript closures and how to use closures in your code more effectively.

Introduction to JavaScript closures

In JavaScript, a closure is a [function](https://www.javascripttutorial.net/javascript-function/) that references variables in the outer scope from its inner scope. The closure preserves the outer scope inside its inner scope.

To understand the closures, you need to know how the lexical scoping works first.

Lexical scoping

Lexical scoping defines the [scope of a variable](https://www.javascripttutorial.net/javascript-variable-scope/) by the position of that variable declared in the source code. For example:

let name = 'John';

function greeting() {

let message = 'Hi';

console.log(message + ' '+ name);

}

Code language: JavaScript (javascript)

In this example:

* The variable name is a global variable. It is accessible from anywhere including within the greeting() function.
* The variable message is a local variable that is accessible only within the greeting() function.

If you try to access the message variable outside the greeting() function, you will get an error.

So the JavaScript engine uses the scope to manage the variable accessibility.

According to lexical scoping, the scopes can be nested and the inner function can access the variables declared in its outer scope. For example:

function greeting() {

let message = 'Hi';

function sayHi() {

console.log(message);

}

sayHi();

}

greeting();

Code language: JavaScript (javascript)

The greeting() function creates a local variable named message and a function named sayHi().

The sayHi() is the inner function that is available only within the body of the greeting() function.

The sayHi() function can access the variables of the outer function such as the message variable of the greeting() function.

Inside the greeting() function, we call the sayHi() function to display the message Hi.

JavaScript closures

Let’s modify the greeting() function:

function greeting() {

let message = 'Hi';

function sayHi() {

console.log(message);

}

return sayHi;

}

let hi = greeting();

hi(); *// still can access the message variable*

Code language: JavaScript (javascript)

Now, instead of executing the sayHi() function inside the greeting() function, the greeting() function returns the sayHi() function object.

Note that functions are the [first-class citizens in JavaScript](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/), therefore, you can return a function from another function.

Outside of the greeting() function, we assigned the hi variable the value returned by the greeting() function, which is a reference of the sayHi() function.

Then we executed the sayHi() function using the reference of that function: hi(). If you run the code, you will get the same effect as the one above.

However, the interesting point here is that, normally, a local variable only exists during the execution of the function.

It means that when the greeting() function has completed executing, the message variable is no longer accessible.

In this case, we execute the hi() function that references the sayHi() function, the message variable still exists.

The magic of this is closure. In other words, the sayHi() function is a closure.

A closure is a function that preserves the outer scope in its inner scope.

More JavaScript Closure example

The following example illustrates a more practical example of closure.

function greeting(message) {

return function(name){

return message + ' ' + name;

}

}

let sayHi = greeting('Hi');

let sayHello = greeting('Hello');

console.log(sayHi('John')); *// Hi John*

console.log(sayHello('John')); *// Hello John*

Code language: JavaScript (javascript)

The greeting() function takes one argument named message and returns a function that accepts a single argument called name.

The return function returns a greeting message that is the combination of the message and name variables.

The greeting() function behaves like a function factory. It creates sayHi() and sayHello() functions with the respective messages Hi and Hello.

The sayHi() and sayHello() are closures. They share the same function body but store different scopes.

In the sayHi() closure, the message is Hi, while in the sayHello() closure the message is Hello.

JavaScript closures in a loop

Consider the following example:

for (var index = 1; index <= 3; index++) {

setTimeout(function () {

console.log('after ' + index + ' second(s):' + index);

}, index \* 1000);

}

Code language: JavaScript (javascript)

Output

after 4 second(s):4

after 4 second(s):4

after 4 second(s):4

Code language: CSS (css)

The code shows the same message.

What we wanted to do in the loop is to copy the value of  i in each iteration at the time of iteration to display a message after 1, 2, and 3 seconds.

The reason you see the same message after 4 seconds is that the callback passed to the setTimeout() a closure. It remembers the value of i from the last iteration of the loop, which is 4.

In addition, all three closures created by the [for-loop](https://www.javascripttutorial.net/javascript-for-loop/) share the same global scope access the same value of i.

To fix this issue, you need to create a new closure scope in each iteration of the loop.

There are two popular solutions: IIFE & let keyword.

1) Using the IIFE solution

In this solution, you use an [immediately invoked function expression](https://www.javascripttutorial.net/javascript-immediately-invoked-function-expression-iife/) (a.k.a IIFE) because an IIFE creates a new scope by declaring a function and immediately execute it.

for (var index = 1; index <= 3; index++) {

(function (index) {

setTimeout(function () {

console.log('after ' + index + ' second(s):' + index);

}, index \* 1000);

})(index);

}

Code language: JavaScript (javascript)

Output

after 1 second(s):1

after 2 second(s):2

after 3 second(s):3

Code language: CSS (css)

2) Using let keyword in ES6

In ES6, you can use the [let](https://www.javascripttutorial.net/javascript-variables/#let) keyword to declare a variable that is block-scoped.

If you use the let keyword in the [for-loop](https://www.javascripttutorial.net/javascript-for-loop/), it will create a new lexical scope in each iteration. In other words, you will have a new index variable in each iteration.

In addition, the new lexical scope is chained up to the previous scope so that the previous value of the index is copied from the previous scope to the new one.

for (let index = 1; index <= 3; index++) {

setTimeout(function () {

console.log('after ' + index + ' second(s):' + index);

}, index \* 1000);

}

Code language: JavaScript (javascript)

Output

after 1 second(s):1

after 2 second(s):2

after 3 second(s):3

Code language: CSS (css)

Summary

* Lexical scoping describes how the JavaScript engine uses the location of the variable in the code to determine where that variable is available.
* A closure is a combination of a function and its ability to remember variables in the outer scope.

# JavaScript Immediately Invoked Function Expression

**Summary**: in this tutorial, you will learn about JavaScript immediately invoked function expressions (IIFE).

## TL;DR

A JavaScript immediately invoked function expression is a [function](https://www.javascripttutorial.net/javascript-function/) defined as an expression and executed immediately after creation. The following shows the syntax of defining an immediately invoked function expression:

(function(){

*//...*

})();

Code language: JavaScript (javascript)

## Why IIFEs

When you define a [function](https://www.javascripttutorial.net/javascript-function/), the JavaScript engine adds the function to the global object. See the following example:

function add(a,b) {

return a + b;

}

Code language: JavaScript (javascript)

In web browsers, the JavaScript engine adds the add() function to the window global object:

console.log(window.add);

Code language: JavaScript (javascript)

Likewise, if you declare a [variable](https://www.javascripttutorial.net/javascript-variables/) outside of a function using the var keyword, the JavaScript engine also adds the variable to the global object:

var counter = 10;

console.log(window.counter); *// 10*

Code language: JavaScript (javascript)

If you have many global variables and functions, the JavaScript engine will only release the memory allocated for them until the global object loses its scopes.

As a result, the script may use the memory inefficiently. On top of that, having global variables and functions will likely cause name collisions.

One way to prevent the functions and variables from polluting the global object is to use immediately invoked function expressions.

In JavaScript, you can have the following expressions:

'This is a string';

(10+20);

Code language: JavaScript (javascript)

This syntax is correct even though the expressions have no effect. A function can be also declared as an expression which is called a function expression:

let sum = function(a, b) {

return a + b;

}

Code language: JavaScript (javascript)

In this syntax, the part on the right side of the assignment operator(=) is a function expression. Because a function is an expression, you can wrap it inside parentheses:

let sum = (function(a, b) {

return a + b;

});

Code language: JavaScript (javascript)

In this example, the sum variable is referenced as the [anonymous function](https://www.javascripttutorial.net/javascript-anonymous-functions/) that adds two arguments.

In addition, you can execute the function immediately after creating it:

let sum = (function(a,b){

return a + b;

})(10, 20);

console.log(sum);

Code language: JavaScript (javascript)

In this example, the sum variable holds the result of the function call.

The following expression is called an immediately invoked function expression (IIFE) because the function is created as an expression and executed immediately:

(function(a,b){

return a + b;

})(10,20);

Code language: JavaScript (javascript)

This is the general syntax for defining an IIFE:

(function(){

*//...*

})();

Code language: JavaScript (javascript)

Note that you can use an [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) to define an IIFE:

(() => {

*//...*

})();

Code language: JavaScript (javascript)

By placing [functions](https://www.javascripttutorial.net/javascript-function/) and [variables](https://www.javascripttutorial.net/javascript-variables/) inside an immediately invoked function expression, you can avoid polluting them to the global object:

(function() {

var counter = 0;

function add(a, b) {

return a + b;

}

console.log(add(10,20)); *// 30*

}());

Code language: JavaScript (javascript)

## Named IIFE

An IIFE can have a name. However, it cannot be invoked again after execution:

(function namedIIFE() {

*//...*

})();

Code language: JavaScript (javascript)

## IIFE starting with a semicolon (;)

Sometimes, you may see an IIFE that starts with a semicolon(;):

;(function() {

*/\* \*/*

})();

Code language: JavaScript (javascript)

In this syntax, the semicolon is used to terminate the statement in case two or more JavaScript files are blindly concatenated into a single file.

For example, you may have two file lib1.js and lib2.js which use IIFEs:

(function(){

*// ...*

})()

(function(){

*// ...*

})()

Code language: JavaScript (javascript)

If you use a code bundler tool to concatenate code from both files into a single file, without the semicolon (;) the concatenated JavaScript code will cause a syntax error.

## IIFE in actions

Suppose that you have a library called calculator.js with the following functions:

function add(a, b) {

return a + b;

}

function mutiply(a, b) {

return a \* b;

}

Code language: JavaScript (javascript)

And you load the calculator.js in an HTML document.

Later, you also want to load another JavaScript library called app.js to the same document:

<!DOCTYPE html>

**<head>**

**<meta charset="UTF-8">**

**<title>**JavaScript IIFE**</title>**

**</head>**

**<body>**

**<script src="calculator.js"></script>**

**<script src="app.js"></script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

The app.js also has the add() function:

function add() {

return 'add';

}

Code language: JavaScript (javascript)

When you use the add() function in the HTML document, it returns the 'add' string instead of the sum of two numbers:

let result = add(10, 20);

console.log(result); *// 'add'*

Code language: JavaScript (javascript)

This is because the add() function in the app.js overrides the add() function in the calculator.js library.

To fix this, you can apply IIFE in the calculator.js as follows:

const calculator = (function () {

function add(a, b) {

return a + b;

}

function multiply(a, b) {

return a \* b;

}

return {

add: add,

multiply: multiply

}

})();

Code language: JavaScript (javascript)

The IIFE returns an object that contains the add and multiply methods that reference the add() and multiply() functions. In the HTML document, you can use the calculator.js library as follows:

<!DOCTYPE html>

**<head>**

**<meta charset="UTF-8">**

**<title>**JavaScript IIFE**</title>**

**</head>**

**<body>**

**<script src="js/calculator.js"></script>**

**<script src="js/app.js"></script>**

**<script>**

let result = calculator.add(10, 20); *// add in app.js*

console.log(result); *// 30*

console.log(add()); *// add in the app.js*

**</script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

The calculator.add() called the add() function exported by the calculator.js while the second call to the add() function references the add() function in the app.js.

## jQuery & IIFE

The following HTML document uses the jQuery library:

<!DOCTYPE html>

**<head>**

**<meta charset="UTF-8">**

**<title>**JavaScript IIFE - jQuery**</title>**

**</head>**

**<body>**

**<h1>**jQuery Demo**</h1>**

**<script src="https://code.jquery.com/jquery-3.4.1.slim.js"**

**integrity="sha256-BTlTdQO9/fascB1drekrDVkaKd9PkwBymMlHOiG+qLI=" crossorigin="anonymous"></script>**

**<script>**

let counter = 1;

$('h1').click(function () {

$(this).text('jQuery Demo' + ' Clicked ' + counter++);

});

**</script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

When you import the jQuery library, you can access many useful jQuery functions via the $ or jQuery object. Under the hood, jQuery uses the IIFE to expose its functionality.

By doing this, jQuery just needs to use one global variable ($) to expose a ton of functions without polluting the global object.

The following example illustrates how to change the jQuery $ object to \_ inside the IIFE:

(function (\_) {

let counter = 1;

\_('h1').click(function () {

\_(this).text('jQuery Demo' + ' Clicked ' + counter++);

});

})(jQuery);

Code language: JavaScript (javascript)

In this example, we passed the jQuery object into the IIFE and used the \_ argument instead.

In this tutorial, you will have learned about the JavaScript immediately invoked function expressions (IIFE) and their purposes.

Returning Multiple Values from a Function

**Summary**: in this tutorial, you will learn to define JavaScript functions that return multiple values.

[JavaScript functions](https://www.javascripttutorial.net/javascript-function/) can return a single value. To return multiple values from a function, you can pack the return values as elements of an [array](https://www.javascripttutorial.net/javascript-array/) or as properties of an [object](https://www.javascripttutorial.net/javascript-objects/).

Returning multiple values from a function using an array

Suppose the following getNames() function retrieves the first name and last name from a database in the backend or from the result of a third-party API call and returns them as elements of an array:

function getNames() {

*// get names from the database or API*

let firstName = 'John',

lastName = 'Doe';

*// return as an array*

return [firstName, lastName];

}

Code language: JavaScript (javascript)

The following shows how to get the return value from the getNames() function:

let names = getNames();

Code language: JavaScript (javascript)

Because the names variable is an array, you can reference its elements using the square brackets, like this:

const firstName = names[0],

lastName = names[1];

Code language: JavaScript (javascript)

In ES6, you can use the [destructuring assignment](https://www.javascripttutorial.net/es6/destructuring/) syntax to unpack values from an array more intuitively, like this:

const [firstName, lastName] = getNames();

Code language: JavaScript (javascript)

In this code, the firstName and lastName variables will take the first and second elements of the return array.

Returning multiple values from an function using an object

If you want to assign a name to each returned value to make it more readable and easier to maintain, you can use an [object](https://www.javascripttutorial.net/javascript-objects/):

function getNames() {

*// get names from the database or API*

let firstName = 'John',

lastName = 'Doe';

*// return values*

return {

'firstName': firstName,

'lastName': lastName

};

}

Code language: JavaScript (javascript)

Since the names of the properties are the same as the variables, you can shorten it using the [object literal syntax extensions in ES6](https://www.javascripttutorial.net/es6/object-literal-extensions/) as follows:

function getNames() {

*// get names from the database or API*

let firstName = 'John',

lastName = 'Doe';

return { firstName, lastName };

}

Code language: JavaScript (javascript)

And you can get the return value as an object like this:

let names = getNames();

let firstName = names.firstName,

lastName = names.lastName;

Code language: JavaScript (javascript)

If you want to unpack properties from an object, you can use the [object destructuring syntax](https://www.javascripttutorial.net/es6/destructuring/) as follows:

let { firstName, lastName } = getNames();

Code language: JavaScript (javascript)

Summary

* JavaScript doesn’t support functions that return multiple values. However, you can wrap multiple values into an array or an object and return the array or the object.
* Use destructuring assignment syntax to unpack values from the array, or properties from objects.

An Introduction to JavaScript Arrow Functions

**Summary**: in this tutorial, you will learn how to use the JavaScript arrow function to write more concise code for function expressions.

Introduction to JavaScript arrow functions

ES6 arrow functions provide you with an alternative way to write a shorter syntax compared to the function expression.

The following example defines a function expression that returns the sum of two numbers:

let add = function (x, y) {

return x + y;

};

console.log(add(10, 20)); *// 30*

Code language: JavaScript (javascript)

The following example is equivalent to the above add() function expression but use an arrow function instead:

let add = (x, y) => x + y;

console.log(add(10, 20)); *// 30;*

Code language: JavaScript (javascript)

In this example, the arrow function has one expression x + y so it returns the result of the expression.

However, if you use the block syntax, you need to specify the return keyword:

let add = (x, y) => { return x + y; };

Code language: JavaScript (javascript)

The  typeof operator returns function indicating the type of arrow function.

console.log(typeof add); *// function*

Code language: JavaScript (javascript)

The arrow function is also an instance of the [Function type](https://www.javascripttutorial.net/javascript-function-type/) as shown in the following example:

console.log(add instanceof Function); *// true*

Code language: JavaScript (javascript)

JavaScript arrow functions with multiple parameters

If an arrow function has two or more parameters, you use the following syntax:

(p1, p2, ..., pn) => expression;

Code language: PHP (php)

The following expression:

=> expression

Code language: PHP (php)

is equivalent to the following expression:

=> { return expression; }

Code language: PHP (php)

For example, to [sort an array](https://www.javascripttutorial.net/javascript-array-sort/) of numbers in the descending order, you use the sort() method of the array object as follows:

let numbers = [4,2,6];

numbers.sort(function(a,b){

return b - a;

});

console.log(numbers); *// [6,4,2]*

Code language: JavaScript (javascript)

The code is more concise with the arrow function syntax:

let numbers = [4,2,6];

numbers.sort((a,b) => b - a);

console.log(numbers); *// [6,4,2]*

Code language: JavaScript (javascript)

JavaScript arrow functions with a single parameter

If an arrow function takes a single parameter, you use the following syntax:

(p1) => { statements }

Code language: PHP (php)

Note that you can omit the parentheses as follows:

p => { statements }

Code language: PHP (php)

The following example uses an arrow function as an argument of the [map()](https://www.javascripttutorial.net/javascript-array-map/) method that transforms an array of strings into an array of the string’s lengths.

let names = ['John', 'Mac', 'Peter'];

let lengths = names.map(name => name.length);

console.log(lengths);

Code language: JavaScript (javascript)

Output:

[ 4, 3, 5 ]

Code language: JSON / JSON with Comments (json)

JavaScript arrow functions with no parameter

If the arrow function has no parameter, you need to use parentheses, like this:

() => { statements }

Code language: PHP (php)

For example:

let logDoc = () => console.log(window.document);

logDoc();

Code language: JavaScript (javascript)

Line break between parameter definition and arrow

JavaScript doesn’t allow you to have a line break between the parameter definition and the arrow (=>) in an arrow function.

For example, the following code causes a SyntaxError:

let multiply = (x,y)

=> x \* y;

Code language: JavaScript (javascript)

However, the following code works perfectly fine:

let multiply = (x,y) =>

x \* y;

Code language: JavaScript (javascript)

JavaScript allows you to have line breaks between parameters as shown in the following example:

let multiply = (

x,

y

) =>

x \* y;

Code language: JavaScript (javascript)

Statements & expressions in the arrow function body

In JavaScript, an expression evaluates to a value as shown in the following example.

10 + 20;

A statement does a specific task such as:

if (x === y) {

console.log('x equals y');

}

Code language: JavaScript (javascript)

If you use an expression in the body of an arrow function, you don’t need to use the curly braces.

let square = x => x \* x;

Code language: JavaScript (javascript)

However, if you use a statement, you must wrap it inside a pair of curly braces as in the following example:

let except = msg => {

throw msg;

};

Code language: JavaScript (javascript)

JavaScript arrow functions and object literal

Consider the following example:

let setColor = function (color) {

return {value: color}

};

let backgroundColor = setColor('Red');

console.log(backgroundColor.value); *// "Red"*

Code language: JavaScript (javascript)

The setColor() function expression returns an object that has the value property set to the color argument.

If you use the following syntax to return an object literal from an arrow function, you will get an error.

p => {object:literal}

Code language: PHP (php)

For example, the following code causes an error.

let setColor = color => {value: color };

Code language: JavaScript (javascript)

Since both block and object literal use curly brackets, the JavasScript engine cannot distinguish between a block and an object.

To fix this, you need to wrap the object literal in parentheses as follows:

let setColor = color => ({value: color });

Code language: JavaScript (javascript)

Arrow function vs. regular function

There are two main differences between an arrow function and a regular function.

1. First, in the arrow function, the [this](https://www.javascripttutorial.net/javascript-this/), arguments, super, [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) are lexical. It means that the arrow function uses these variables (or constructs) from the enclosing lexical scope.
2. Second, an arrow function cannot be used as a function constructor. If you use the new keyword to create a new object from an arrow function, you will get an error.

JavaScript arrow functions and this value

In JavaScript, a new function defines its own this value. However, it is not the case for the arrow function. See the following example:

function Car() {

this.speed = 0;

this.speedUp = function (speed) {

this.speed = speed;

setTimeout(function () {

console.log(this.speed); *// undefined*

}, 1000);

};

}

let car = new Car();

car.speedUp(50);

Code language: JavaScript (javascript)

Inside the anonymous function of the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function, the this.speed is undefined. The reason is that the this of the [anonymous function](https://www.javascripttutorial.net/javascript-anonymous-functions/) shadows the this of the speedUp() method.

To fix this, you assign the this value to a variable that doesn’t shadow inside the anonymous function as follows:

function Car() {

this.speed = 0;

this.speedUp = function (speed) {

this.speed = speed;

let self = this;

setTimeout(function () {

console.log(self.speed);

}, 1000);

};

}

let car = new Car();

car.speedUp(50); *// 50;*

Code language: JavaScript (javascript)

Unlike an anonymous function, an arrow function captures the this value of the enclosing context instead of creating its own this context. The following code should work as expected:

function Car() {

this.speed = 0;

this.speedUp = function (speed) {

this.speed = speed;

setTimeout(

() => console.log(this.speed),

1000);

};

}

let car = new Car();

car.speedUp(50); *// 50;*

Code language: JavaScript (javascript)

JavaScript arrow functions and the arguments object

An arrow function doesn’t have the arguments object. For example:

function show() {

return x => x + arguments[0];

}

let display = show(10, 20);

let result = display(5);

console.log(result); *// 15*

Code language: JavaScript (javascript)

The arrow function inside the showMe() function references the arguments object. However, this arguments object belongs to the show() function, not the arrow function.

Also, an arrow function doesn’t have the [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) keyword.

JavaScript arrow functions and the prototype property

When you define a [function](https://www.javascripttutorial.net/javascript-function/) using a function keyword, the function has a property called [prototype](https://www.javascripttutorial.net/javascript-prototype/):

function dump( message ) {

console.log(message);

}

console.log(dump.hasOwnProperty('prototype')); *// true*

Code language: JavaScript (javascript)

However, arrow functions don’t have the prototype property:

let dump = message => console.log(message);

console.log(dump.hasOwnProperty('prototype')); *// false*

Code language: JavaScript (javascript)

It is a good practice to use arrow functions for [callbacks](https://www.javascripttutorial.net/javascript-callback/) and [closures](https://www.javascripttutorial.net/javascript-closure/) because the syntax of arrow functions is cleaner.

Summary

* Use the (...args) => expression; to define an arrow function.
* Use the (...args) => { statements } to define an arrow function that has multiple statements.
* An arrow function doesn’t have its binding to this or super.
* An arrow function doesn’t have arguments object, new.target keyword, and prototype property.

When You Should Not Use Arrow Functions

**Summary**: in this tutorial, you will learn when you **should not use** the arrow functions in ES6.

An [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) doesn’t have its own [this](https://www.javascripttutorial.net/javascript-this/) value and the arguments object. Therefore, you should not use it as an event handler, a method of an object literal, a prototype method, or when you have a function that uses the arguments object.

1) Event handlers

Suppose that you have the following input text field:

**<input type="text" name="username" id="username" placeholder="Enter a username">**

Code language: HTML, XML (xml)

And you want to show a greeting message when users type their usernames. The following shows the <div> element that will display the greeting message:

**<div id="greeting"></div>**

Code language: HTML, XML (xml)

Once users type their usernames, you capture the current value of the input and update it to the <div> element:

const greeting = document.querySelector('#greeting');

const username = document.querySelector('#username');

username.addEventListener('keyup', () => {

greeting.textContent = 'Hello ' + this.value;

});

Code language: JavaScript (javascript)

However, when you execute the code, you will get the following message regardless of whatever you type:

Hello undefined

Code language: JavaScript (javascript)

It means that the this.value in the event handler always returns undefined.

As mentioned earlier, the arrow function doesn’t have its own this value. It uses the this value of the enclosing lexical scope. In the above example, the this in arrow function references the global object.

In the web browser, the global object is window. The window object doesn’t have the value property. Therefore, the JavaScript engine adds the value property to the window object and sets its values to undefined.

To fix this issue, you need to use a regular function instead. The this value will be bound to the <input> element that triggers the event.

username.addEventListener('keyup', function () {

input.textContent = 'Hello ' + this.value;

});

Code language: JavaScript (javascript)

2) Object methods

See the following counter object:

const counter = {

count: 0,

next: () => ++this.count,

current: () => this.count

};

Code language: JavaScript (javascript)

The counter object has two methods: current() and next(). The current() method returns the current counter value and the next() method returns the next counter value.

The following shows the next counter value which should be 1:

console.log(counter.next());

Code language: CSS (css)

However, it returns NaN.

The reason is that when you use the arrow function inside the object, it inherits the this value from the enclosing lexical scope which is the global scope in this example.

The this.count inside the next() method is equivalent to the window.count (in the web browser).

The window.count is undefined by default because the window object doesn’t have the count property. The next() method adds one to undefined that results in NaN.

To fix this, you use regular functions as the method of an object literal as follows:

const counter = {

count: 0,

next() {

return ++this.count;

},

current() {

return this.count;

}

};

Code language: JavaScript (javascript)

Now, calling the next() method will return one as expected:

console.log(counter.next()); *// 1*

Code language: JavaScript (javascript)

3) Prototype methods

See the following Counter object that uses the [prototype](https://www.javascripttutorial.net/javascript-prototype/) pattern:

function Counter() {

this.count = 0;

}

Counter.prototype.next = () => {

return this.count;

};

Counter.prototype.current = () => {

return ++this.next;

}

Code language: JavaScript (javascript)

The this value in these next() and current() methods reference the global object. Since you want the this value inside the methods to reference the Counter object, you need to use the regular functions instead:

function Counter() {

this.count = 0;

}

Counter.prototype.next = function () {

return this.count;

};

Counter.prototype.current = function () {

return ++this.next;

}

Code language: JavaScript (javascript)

4) Functions that use the arguments object

Arrow functions don’t have the arguments object. Therefore, if you have a function that uses arguments object, you cannot use the arrow function.

For example, the following concat() function won’t work:

const concat = (separator) => {

let args = Array.prototype.slice.call(arguments, 1);

return args.join(separator);

}

Code language: JavaScript (javascript)

Instead, you use a regular function like this:

function concat(separator) {

let args = Array.prototype.slice.call(arguments, 1);

return args.join(separator);

}

Code language: JavaScript (javascript)

Summary

* An arrow function doesn’t have its own this value. Instead, it uses the this value of the enclosing lexical scope. An arrow function also doesn’t have the arguments object.
* Avoid using the arrow function for event handlers, object methods, prototype methods, and functions that use the arguments object.

# JavaScript Rest Parameters

**Summary**: in this tutorial, you will learn how to use the JavaScript rest parameters to gather parameters and put them all in an array.

## Introduction to JavaScript rest parameters

ES6 provides a new kind of parameter so-called rest parameter that has a prefix of three dots (...). A rest parameter allows you to represent an indefinite number of arguments as an [array](https://www.javascripttutorial.net/javascript-array/). See the following syntax:

function fn(a,b,...args) {

*//...*

}

Code language: JavaScript (javascript)

The last parameter (args) is prefixed with the three-dots ( ...). It’s called a rest parameter ( ...args).

All the arguments you pass to the [function](https://www.javascripttutorial.net/javascript-function/) will map to the parameter list. In the syntax above, the first argument maps to a, the second one maps to b, and the third, the fourth, etc., will be stored in the rest parameter args as an array. For example:

fn(1, 2, 3, "A", "B", "C");

Code language: JavaScript (javascript)

The args array stores the following values:

[3,'A','B','C']

Code language: JSON / JSON with Comments (json)

If you pass only the first two parameters, the rest parameter will be an empty array:

fn(1,2);

The args will be:

[]

Code language: JSON / JSON with Comments (json)

Notice that the rest parameters must appear at the end of the argument list. The following code will result in an error:

function fn(a,...rest, b) {

*// error*

}

Code language: JavaScript (javascript)

Error:

SyntaxError: Rest parameter must be last formal parameter

Code language: JavaScript (javascript)

## More JavaScript rest parameters examples

See the following example:

function sum(...args) {

let total = 0;

for (const a of args) {

total += a;

}

return total;

}

sum(1, 2, 3);

Code language: JavaScript (javascript)

The output of the script is:

6

In this example, args in an array. Therefore, you could use the [for..of](https://www.javascripttutorial.net/es6/javascript-for-of/) loop to iterate over its elements and sum them up.

Assuming that the caller of the sum() function may pass arguments with various kinds of data types such as [number](https://www.javascripttutorial.net/javascript-data-types/#number), [string](https://www.javascripttutorial.net/javascript-data-types/#string), and [boolean](https://www.javascripttutorial.net/javascript-data-types/" \l "boolean), and you want to calculate the total of numbers only:

function sum(...args) {

return args

.filter(function (e) {

return typeof e === 'number';

})

.reduce(function (prev, curr) {

return prev + curr;

});

}

Code language: JavaScript (javascript)

The following script uses the new sum() function to sum only numeric arguments:

let result = sum(10,'Hi',null,undefined,20);

console.log(result);

Code language: JavaScript (javascript)

Output:

30

Note that without the rest parameters, you have to use the arguments object of the function.

However, the arguments object itself is not an instance of the Array type. Therefore, you cannot use the filter() method directly. In ES5, you have to use Array.prototype.filter.call() as follows:

function sum() {

return Array.prototype.filter

.call(arguments, function (e) {

return typeof e === 'number';

})

.reduce(function (prev, curr) {

return prev + curr;

});

}

Code language: JavaScript (javascript)

As you see, the rest parameter makes the code more elegant. Suppose you need to filter the arguments based on a specific type such as numbers, strings, boolean, and null. The following function helps you to do it:

function filterBy(type, ...args) {

return args.filter(function (e) {

return typeof e === type;

});

}

Code language: JavaScript (javascript)

## JavaScript rest parameters and arrow function

An [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) does not have the arguments object. Therefore, if you want to pass some arguments to the arrow function, you must use the rest parameters. See the following example:

const combine = (...args) => {

return args.reduce(function (prev, curr) {

return prev + ' ' + curr;

});

};

let message = combine('JavaScript', 'Rest', 'Parameters'); *// =>*

console.log(message); *// JavaScript Rest Parameters*

Code language: JavaScript (javascript)

Output:

JavaScript Rest Parameters

The combine() function is an arrow that takes an indefinite number of arguments and concatenates these arguments.

## JavaScript rest parameter in a dynamic function

JavaScript allows you to create dynamic functions through the [Function](https://www.javascripttutorial.net/javascript-function-type/) constructor. And it is possible to use the rest parameter in a dynamic function. Here is an example:

var showNumbers = new Function('...numbers', 'console.log(numbers)');

showNumbers(1, 2, 3);

Code language: PHP (php)

Output:

[ 1, 2, 3 ]

Code language: JSON / JSON with Comments (json)

In this tutorial, you have learned how to use the JavaScript rest parameter to represent an indefinite number of arguments as an array.

JavaScript Callbacks

**Summary**: in this tutorial, you will learn about JavaScript callback functions including synchronous and asynchronous callbacks.

What are callbacks

In JavaScript, [functions are first-class citizens](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/). Therefore, you can pass a [function](https://www.javascripttutorial.net/javascript-function/) to another function as an argument.

By definition, a callback is a [function](https://www.javascripttutorial.net/javascript-function/) that you pass into another function as an argument for executing later.

The following defines a filter() function that accepts an [array](https://www.javascripttutorial.net/array/) of numbers and returns a new array of odd numbers:

function filter(numbers) {

let results = [];

for (const number of numbers) {

if (number % 2 != 0) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers));

Code language: JavaScript (javascript)

How it works.

* First, define the filter() function that accepts an array of numbers and returns a new array of the odd numbers.
* Second, define the numbers array that has both odd and even numbers.
* Third, call the filter() function to get the odd numbers out of the numbers array and output the result.

If you want to return an array that contains even numbers, you need to modify the filter() function. To make the filter() function more generic and reusable, you can:

* First, extract the logic in the if block and wrap it in a separate function.
* Second, pass the function to the filter() function as an argument.

Here’s the updated code:

function isOdd(number) {

return number % 2 != 0;

}

function filter(numbers, fn) {

let results = [];

for (const number of numbers) {

if (fn(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers, isOdd));

Code language: JavaScript (javascript)

The result is the same. However, you can pass any function that accepts an argument and returns a boolean value to the second argument of the filter() function.

For example, you can use the filter() function to return an array of even numbers like this:

function isOdd(number) {

return number % 2 != 0;

}

function isEven(number) {

return number % 2 == 0;

}

function filter(numbers, fn) {

let results = [];

for (const number of numbers) {

if (fn(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers, isOdd));

console.log(filter(numbers, isEven));

Code language: JavaScript (javascript)

By definition, the isOdd and isEven are callback functions or callbacks. Because the filter() function accepts a function as an argument, it’s called a *high-order function*.

A callback can be an anonymous function, which is a function without a name like this:

function filter(numbers, callback) {

let results = [];

for (const number of numbers) {

if (callback(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, function (number) {

return number % 2 != 0;

});

console.log(oddNumbers);

Code language: JavaScript (javascript)

In this example, we pass an anonymous function to the filter() function instead of using a separate function.

In ES6, you can use an [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) like this:

function filter(numbers, callback) {

let results = [];

for (const number of numbers) {

if (callback(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, (number) => number % 2 != 0);

console.log(oddNumbers);

Code language: JavaScript (javascript)

There are two types of callbacks: synchronous and asynchronous callbacks.

Synchronous callbacks

A synchronous callback is executed during the execution of the high-order function that uses the callback. The isOdd and isEven are examples of synchronous callbacks because they execute during the execution of the filter() function.

Asynchronous callbacks

An asynchronous callback is executed after the execution of the high-order function that uses the callback.

Asynchronicity means that if JavaScript has to wait for an operation to complete, it will execute the rest of the code while waiting.

Note that JavaScript is a single-threaded programming language. It carries asynchronous operations via the callback queue and[event loop](https://www.javascripttutorial.net/javascript-event-loop/).

Suppose that you need to develop a script that downloads a picture from a remote server and process it after the download completes:

function download(url) {

*// ...*

}

function process(picture) {

*// ...*

}

download(url);

process(picture);

Code language: JavaScript (javascript)

However, downloading a picture from a remote server takes time depending on the network speed and the size of the picture.

The following download() function uses the setTimeout() function to simulate the network request:

function download(url) {

setTimeout(() => {

*// script to download the picture here*

console.log(`Downloading ${url} ...`);

},1000);

}

Code language: JavaScript (javascript)

And this code emulates the process() function:

function process(picture) {

console.log(`Processing ${picture}`);

}

Code language: JavaScript (javascript)

When you execute the following code:

let url = 'https://www.javascripttutorial.net/pic.jpg';

download(url);

process(url);

Code language: JavaScript (javascript)

you will get the following output:

Processing https:*//javascripttutorial.net/pic.jpg*

Downloading https:*//javascripttutorial.net/pic.jpg ...*

Code language: JavaScript (javascript)

This is not what you expected because the process() function executes before the download() function. The correct sequence should be:

* Download the picture and wait for the download completes.
* Process the picture.

To resolve this issue, you can pass the process() function to the download() function and execute the process() function inside the download() function once the download completes, like this:

function download(url, callback) {

setTimeout(() => {

*// script to download the picture here*

console.log(`Downloading ${url} ...`);

*// process the picture once it is completed*

callback(url);

}, 1000);

}

function process(picture) {

console.log(`Processing ${picture}`);

}

let url = 'https://wwww.javascripttutorial.net/pic.jpg';

download(url, process);

Code language: JavaScript (javascript)

Output:

Downloading https:*//www.javascripttutorial.net/pic.jpg ...*

Processing https:*//www.javascripttutorial.net/pic.jpg*

Code language: JavaScript (javascript)

Now, it works as expected.

In this example, the process() is a callback passed into an asynchronous function.

When you use a callback to continue code execution after an asynchronous operation, the callback is called an asynchronous callback.

To make the code more concise, you can define the process() function as an anonymous function:

function download(url, callback) {

setTimeout(() => {

*// script to download the picture here*

console.log(`Downloading ${url} ...`);

*// process the picture once it is completed*

callback(url);

}, 1000);

}

let url = 'https://www.javascripttutorial.net/pic.jpg';

download(url, function(picture) {

console.log(`Processing ${picture}`);

});

Code language: JavaScript (javascript)

Handling errors

The download() function assumes that everything works fine and does not consider any exceptions. The following code introduces two callbacks: success and failure to handle the success and failure cases respectively:

function download(url, success, failure) {

setTimeout(() => {

console.log(`Downloading the picture from ${url} ...`);

!url ? failure(url) : success(url);

}, 1000);

}

download(

'',

(url) => console.log(`Processing the picture ${url}`),

(url) => console.log(`The '${url}' is not valid`)

);

Code language: JavaScript (javascript)

Nesting callbacks and the Pyramid of Doom

How do you download three pictures and process them sequentially? A typical approach is to call the download() function inside the callback function, like this:

function download(url, callback) {

setTimeout(() => {

console.log(`Downloading ${url} ...`);

callback(url);

}, 1000);

}

const url1 = 'https://www.javascripttutorial.net/pic1.jpg';

const url2 = 'https://www.javascripttutorial.net/pic2.jpg';

const url3 = 'https://www.javascripttutorial.net/pic3.jpg';

download(url1, function (url) {

console.log(`Processing ${url}`);

download(url2, function (url) {

console.log(`Processing ${url}`);

download(url3, function (url) {

console.log(`Processing ${url}`);

});

});

});

Code language: JavaScript (javascript)

Output:

Downloading https:*//www.javascripttutorial.net/pic1.jpg ...*

Processing https:*//www.javascripttutorial.net/pic1.jpg*

Downloading https:*//www.javascripttutorial.net/pic2.jpg ...*

Processing https:*//www.javascripttutorial.net/pic2.jpg*

Downloading https:*//www.javascripttutorial.net/pic3.jpg ...*

Processing https:*//www.javascripttutorial.net/pic3.jpg*

Code language: JavaScript (javascript)

The script works perfectly fine.

However, this callback strategy does not scale well when the complexity grows significantly.

Nesting many asynchronous functions inside callbacks is known as the **pyramid of doom** or the **callback hell**:

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

....

});

});

});

});

});

Code language: JavaScript (javascript)

To avoid the pyramid of doom, you use [promises](https://www.javascripttutorial.net/es6/javascript-promises/) or [async/await](https://www.javascripttutorial.net/es-next/javascript-async-await/) functions.

Summary

* A callback is a function passed into another function as an argument to be executed later.
* A high-order function is a function that accepts another function as an argument.
* Callback functions can be synchronous or asynchronous.